AN INTERACTIVE, INCREMENTAL ASSEMBLY LANGUAGE PROCESSOR FOR THE INTEL 8080

John L. Cuzzocrea

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

An Interactive, Incremental Assembly Language Processor for the INTEL 8080

by

John L. Cuzzocrea
and
Michael Charles Thomas

June 1977

Thesis Advisor: V. Michael Powers

Approved for public release; distribution unlimited

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) An Interactive, Incremental Assembly Language Processor for the INTEL 8080		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis; June 1977 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) John L.Cuzzocrea Michael Charles Thomas		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE June 1977 13. NUMBER OF PAGES 73
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Naval Postgraduate School Monterey, California 93940		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Incremental compilation
Assembler
Interactive assembler

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The design and implementation of an interactive, incremental assembly system on an INTEL 8080-based microcomputer has been described. Instead of requiring separate editing, assembling and debugging steps, the system allows entry, translation and error checking simultaneously. The implementation is comprised of an integrated set of modules which

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601 |
(Page 1)

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

assemble and execute the source code. The design goals, solutions, and recommendations for further expansion of the system have been presented. The system was implemented in PL/M for use in a diskette-based environment.

Approved for public release; distribution unlimited

AN INTERACTIVE, INCREMENTAL ASSEMBLY LANGUAGE PROCESSOR FOR THE INTEL 8080

by

John L. Cuzzocrea
Lieutenant, United States Navy
B.S., Oklahoma University, 1971

Michael Charles Thomas
Lieutenant, United States Navy
B.S., Oregon State University, 1970

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1977

ABSTRACT

The design and implementation of an interactive, incremental assembly system on an INTEL 8080-based microcomputer has been described. Instead of requiring separate editing, assembling and debugging steps, the system allows entry , translation and error checking simultaneously. The implementation is comprised of an integrated set of modules which assemble and execute the source code. The design goals, sclutions, and recommendations for further expansion of the system have been presented. The system was implemented in PL/M for use in a diskette-based environment.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

I. INTRODUCTION

The software development process typically has been a sequence of three steps. First the user would create his program on some machine readable medium. Secondly he would assemble or compile his source code, going back to step one as needed to correct syntax errors until a successful assembly or compilation was achieved. The third step would involve debugging and running the resultant object code from the assembly or compilation. Of course, if there were any program errors detected in the final step the user would have to retreat to step one and repeat the entire process until he was satisfied that his program would perform in the manner for which it was created.

During this typical three step process, the earliest point at which the user had any feedback from the computer as to the correctness of the syntax of his source code was at completion of the first attempt at compilation or assembly. The user had no idea at all as to how his program would execute until a complete assembly or compilation had been achieved.

Assembly System for Interactive Development (ASID) was an attempt to demonstrate that the user could begin to receive information regarding his source program at the earliest practical point. Not only could the user receive immediate feedback concerning the correctness of the syntax of each program sentence, but it has been shown that he can receive much helpful information concerning the logical construction of his program at the same time.

II. DECREASING SOFTWARE DEVELOPMENT TIME

A. PHYSIOLOGICAL WEAKNESSES OF THE HUMAN BEING

The designer of an effective man-machine interface must be aware of the basic weaknesses of the human physiology as it affects the man-machine relationship. The designer should also account for the differences in experience levels among the user population.

The computer, with its capability for exact recall over potentially infinite periods of time, should be used to aid the user in reconstructing events which he can not recall precisely. Another important factor is anxiety in the user. The machine must respond with some sort of signal, either audio or visual, to reassure the user that it is working for him. Without this periodic reassurance, the user can become perplexed and lose his train of thought.

User attitude can also affect the man-machine interface. Flexibility in the interface, allowing the experienced user to take shortcuts or providing optional verbosity for the inexperienced user, can help to promote a more relaxed atmosphere which is conducive to the high level of concentration needed to develop and debug programs.

B. DEBUGGING TIME

Once the user has established an interface with the machine and begins to use the computer to perform tasks, the machine should aid the user during all phases of the software development process. If the user determines that the tasks given the machine were not performed properly or the machine determines that it does not know what the task given to it means, then the computer should respond with a set of helpful notices. That is, the machine must aid the user in determining why the specific task was not performed or why the task was not performed properly. This function is collectively called debugging and separates into two sections. The first portion is concerned with eliminating assembly/compile time errors or syntax errors. The computer is generally quite proficient at detecting and alerting the user to misuses of the input language.

The second part of debugging has to do with run-time or "logic" errors. These errors manifest themselves as unexpected results from execution of the object program. The machine is not proficient at locating errors of this type unless the machine or operating system is put into an abnormal state, i.e. attempting to execute a data area. Controlled execution monitors (debuggers) are the most effective tools with which to confront logic or run-time errors.

Some of the most common run-time errors that occur are grouped into the following categories: 1) initialization, 2) addressing, 3) referencing, 4) counting and calculating, 5) masking and comparing, 6) estimation of the range of limits, 7) ordering of code [Ref. 1].

III. THE COMPUTER AS AN AID TO SOFTWARE DEVELOPMENT

A. INTERACTIVE MAN MACHINE INTERFACE

If a user is to make efficient use of his time while utilizing a computer system, the system must be designed with the needs of the user in mind. Perhaps one of the best statements of the recognition that considerable attention should be given to the user comes from Dr. James Martin of the IBM Systems Research Institute:

> Increasingly..., man must become the prime focus of the system design. The computer is there to serve him, to obtain information for him and to help him do his job. The ease with which he communicates with it will determine the extent to which he uses it. Whether or not he uses it powerfully will depend upon the man-machine language available to him and how well he is able to understand it."

Thus a system should interact with the user. But how does one discover the "best" method for designing an interactive system? Can the user simply be asked what he would like to have happen when he sits down at a terminal? Apparently not, according to several recent writers on the subject. Further, quite contrary to popular opinion, "armchair" intuitive design techniques have not provided a sufficient basis for system designers to use [Ref. 2]. One approach is to allow the interface between the man and the machine to be alterable by the user under operating conditions without the necessity for reprogramming the system [Ref. 2]. Therefore the user interface is originally coded with the capability for making differential

responses tc a variety of users under a wide range of conditions. This gives tne interactive system interface flexibility. The question then becomes how much flexibility is required by the user.

1. History

There are three kinds of interactive high-level language systems: 1) interactive compilation systems, 2) interactive interpretation systems, 3) interactive direct execution systems [Ref. 3]. The general nature of these categories of interactive systems is shown in Table I.

An interactive compilation system is an interactive high-level language system in which a compiler is employed. Table I lists three types. The type 1(a) system allows the source code to be input at a terminal and a text editcr is available for making changes and corrections. After the entire source is entered, the compiler is called to translate the source code into a block of machine code. During compilation, the syntax of the source code is checked and the syntax error messages are later printed out. If compilation is successful, the machine code is loaded into memory and executed. This entire process is repeated until the program is completely debugged of syntax or compilation time errcrs. Note that the level of interaction is limited such that the user must submit his entire program to the translator before he receives any feedback.

TABLE I

Types of Interactive High-Level Language Systems

1. Interactive Compilation Systems

   Type 1(a): inputing and text editing the entire source code,
   compiling and syntax checking the entire source code,
   executing the object code.

   Type 1(b): inputing and text editing the entire source code,
   syntax checking the entire source code,
   compiling the entire source code,
   executing the object code.

   Type 1(c): inputing and text editing and syntax checking each line of source code.
   compiling the accumulated source code,
   executing the accumulated source code.

2. Interactive Interpretation Systems

   Type 2(a): inputing and editing the entire source code,
   interpreting and syntax checking the entire source code.

   Type 2(b): inputing and text editing each line of source code,
   interpreting and syntax checking a line of source code,

3. Interactive Direct-Execution Systems

   Type 3(a): inputing each symbol of source code,
   syntax checking and executing the symbol,
   text editing the symbol and the code if in error.

The type 1(b) system is similar to the type 1(a) system except that the type 1(b) employs a syntax interpreter for syntax checking before compilation. In this system syntax errors could be detected by the computer and corrected by the user prior to the first attempt at compilation. In some compilers syntax checking is accomplished as the first pass of the compilation. This is an improvement in the amount of interaction allowed of the user because most syntax errors would be found prior to calling in the compiler. The entire program creation process should require less time. However, the user has no opportunity to interact with regard to syntax error correction until after he has typed in his entire program and started the syntax checker.

The type 1(c) system interacts with the user at the level of one line. As each line of the source is entered, it is syntax checked and then put into the text file. Whenever the user wishes to execute the source code in the text file, the source code is then compiled, linked and executed. Alternatively, each line of the source code could be entered, syntax checked and compiled, and then placed in the text file.

An interactive interpretation system is an interactive high level language system in which an interpreter is employed. The high-level language is the programming language. There is neither compilation nor assembly. The user writes only high-level language programs.

Two types are shown in Table I. The type 2(a) system allows the source code to be entered on the terminal and a text editor is available for making changes and corrections as the source code is being typed on the terminal. After

the entire source code is typed, the interpreter is called to check syntax and interpret the source. It is conceptually simple, but the unit of interaction with this type of system is again the entire source code.

The type 2(b) system is similar to the type 2(a) system except that entering, syntax checking and interpreting are carried out one line at a time. As a result this system provides more interaction between the user and the system.

An interactive direct execution system is an interactive high level language system in which a direct execution interpreter is employed. As indicated in Table I, as each symbol of the source code is being entered at the terminal, the symbol is syntax checked and executed. This is accomplished by the "interpretive direct execution loop." An error message is printed out when an error is detected. There would be no "error snowballing" as could happen during a compilation run, whereby the compiler erroneously detects errors in following statements that are in fact syntactically correct. This process of symbol-by-symbol typing, checking and execution gives a maximum interaction between the user and the system.

The interactive direct-execution system also assists the user in debugging logical errors by showing the partial result whenever it is requested. It also allows the user to command the system to execute the accumulated source code from the beginning and to display the result at specified places. When the source code is completely entered, it could have already run once and could have been partially debugged. As with the interactive interpreter system, the user writes only high level language software. The system could be designed so that once the source code is debugged, it could then be run without any further syntax checking in

order to speed up the execution. It is conceivable that the system could be designed so that it serves as a means for one to learn the high level language after a minimum amount of reading or instruction.

2. Incremental Translation and Execution as Aids to Debugging

Referring to the conventional three-step software development process mentioned in the introduction, the authors suggest that the syntax checking process and the first pass of the assembly or compilation could easily be accomplished concurrently with step one, initial program input. The savings in time prior to completing the first successful assembly or compilation should be significant. Snowballing or cascading of syntax errors as is so common in many language processors could be all but eliminated. If the user were utilizing a dedicated computer hardware system such as one of the increasingly popular microcomputer development systems, then the CPU would be asked to perform more work in the same time frame than when only a text editor is used for program creation.

The most obvious improvement is that the user has continuous assurance that the work he has done is indeed syntactically correct. If an error is detected it is corrected before proceeding.

A much more powerful extension of incremental processing is to execute each executable segment of the program as it is successfully parsed. Obviously some restrictions would have to be made on this incremental execution. A call to a non-existant subroutine would not be allowed. The logic that was to determine whether the subroutine was to be called or not could be verified,

though. The display of intermediate results at the termination of each incremental execution would give the user a fairly detailed view of the logical flow of his program. All of this is still happening at step one of the conventional development cycle.

3. State Of The Art Approaches

a. CAPS

An example of an interactive diagnostic compiler-interpreter system is CAPS which is in use at the University of Illinois at Urbana-Champaign [Ref. 4]. It allows beginning programmers to prepare, debug and execute fairly simple programs at a graphics display terminal. Complete syntax checking and most semantic analysis is performed as the program is entered and as it is subsequently edited. Analysis by the system is performed character by character. A remarkable feature of CAPS is its ability to automatically diagnose errors both at compile time and at run time. Errors are not automatically corrected. Instead, CAPS interacts with the user to help him find the cause of the error. Most of the components of CAPS are table driven, both to reduce the space needed for implementation and to increase the flexibility of the system. CAPS supports the beginning programmer who is using either Fortran, PL/I or Cobol.

The principle modules of the CAPS system are a program editor, a syntactic and static semantic error diagnostician, an interpreter for each language supported, a run time error analyzer, a user program file manager and a system table builder and a file manager (Figure 1). Control of the system is distributed throughout the modules. The

user, however, is never aware of this modularity and never has to remember command syntax because each time the system is ready for a command the module that will interpret the command displays a menu of possible actions (Figure 2).

In CAPS, the interactive debugging session is directed by the system and not by the user. This is essential because the beginning programmer does not know what questions to ask; he does not know how to debug. An added benefit of this is that the user does not have to learn a command language for the debugging package.

Currently, since CAPS uses the Plato IV system and has severe time and space constraints imposed on it by the Plato IV system, its capabilities are limited, and it has been only a qualified success. Over 500 people have used CAPS while learning Fortran and PL/I. The diagnostic assistance in the interactive environment is clearly superior to any batch system or interactive system for the beginning programmer. The problem that causes CAPS to be only a qualified success is the time sharing system in which it operates. When Plato IV is handling 500 people simultaneously, even if only 30 terminals run CAPS, the user gets frustratingly slow responce - slow, even for the "hunt and peck" typist writing in an unfamiliar language.

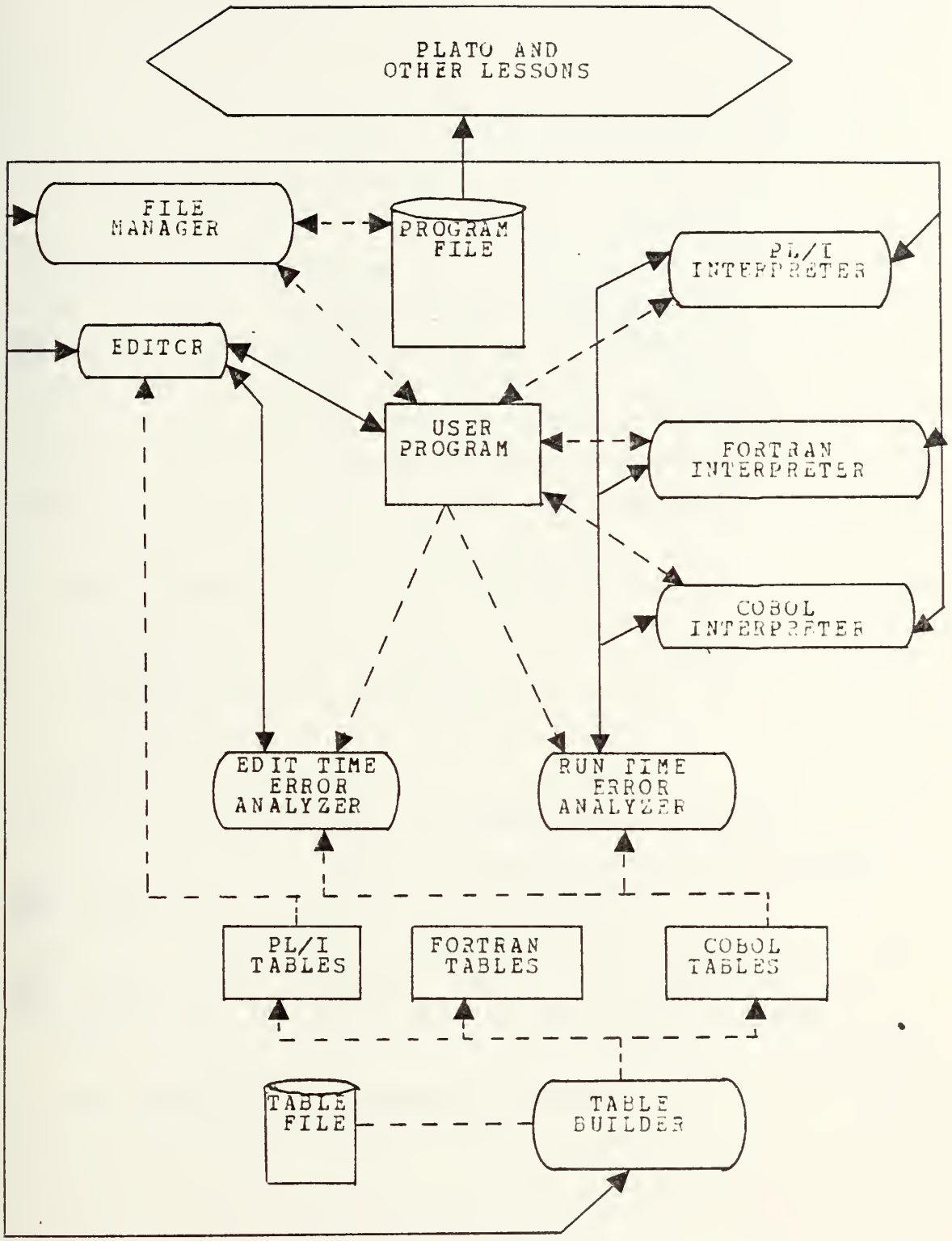

Figure 1 - CAPS system organization

```
      YOUR WORKSPACE IS EMPTY

   PRESS...          TO...
  NEXT               WRITE A COBOL PROGRAM
  (*) NEXT           WRITE IN ANOTHER LANGUAGE
  EDIT               EDIT ONE OF YOUR OLD PROGRAMS
  ERASE              ERASE ONE OF YOUR OLD PROGRAMS
   LAB               EXECUTE ONE OF YOUR PROGRAMS
  DATA               SEE A LIST OF PROGRAMS ON FILE

A) BEFORE STARTING TO WRITE A PPROGRAM

      WORKSPACE CONTAINS TEXT
      (PRESS DATA FOR DETAILS)

   PRESS...          TO...
  NEXT               EDIT YOUR WORKSPACE SOME MORE
   LAB                EXECUTE YOUR WORKSPACE AS A PROGRAM
  (*) BACK           CLEAR YOUR WORKSPACE
  COPY               COPY WORKSPACE INTO A FILE
  (*) COPY           REPLACE 'COBOLK' WITH WORKSPACE
  ERASE              ERASE FILE 'COBOLK' FROM THE FILE

B) AFTER EDITING "CLD PROGRAM" 'COLBOLK'
```

Figure 2 - Typical display of possible actions.

b. IHLLDE

IHLLDE is an example of an interactive direct execution system. It accepts a subset of Algol 60 [Ref. 3].

The system configuration of IHLLDE is shown in Figure 3. There are seven system units: the monitor, input processor, direct execution interpreter, text editor, scanner, I/O processor and teletype. The monitor controls the operation of all system units directly or indirectly. All of the inputs from and the outputs to the teletype are handled by the I/O processor. The scanner is called by the interpreter only. The monitor operates in four modes: monitor mode, input mode, edit mode and run mode.

The system has been implemented on the Univac 1108 computer at the University of Maryland. System operation can best be described by means of an example terminal session.The teletype output is shown in Figure 4 where the user's input items are indicated by underscoring. The user began his session by typing "I" to the monitor to enter the input mode. Then the user proceeded to enter his program at the terminal. The user misspelled "INTEGER", and the system responded by printing an asterisk under the offending symbol together with an error message. The user retyped the line starting with the symbol in error. The user next entered two "READ" statements; the system requested the data by showing "DATA?" after each "READ" statement, because the "READ" statement had immediately been executed. The user next mistyped an assignment statement and he, after the system responded, started the correction from the symbol in error. The user next mistyped the "WRITE" statment twice and then corrected it. The corrected "WRITE" statement was immediately executed and the value of

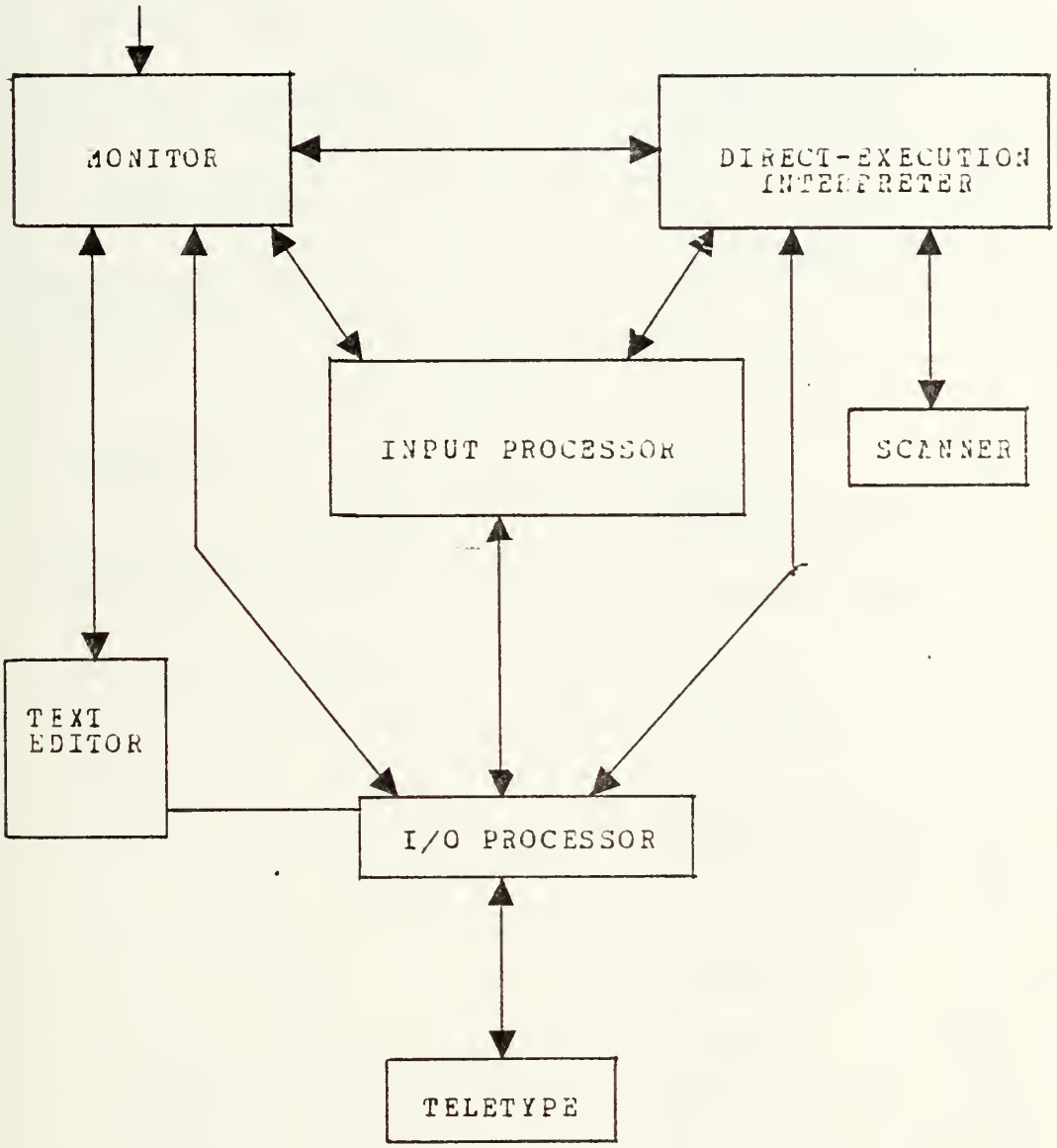

Figure 3 - Configuration of an interactive direct-execution high-level language system.

```
I
INPUT
$BEGIN
INTEGRE A,B;
*
MISSING "INTEGER"

INTEGER A,B;

READ(B);
DATA?
-
READ(A);
DATA?
4
A=A+B;
 *
MISSING ":="

:=A+B;

WRIT(A);

UNDECLARED ID
WRITE(B9;
      *

UNDECLARED ID
A
A = 7

);
END $

PROGRAM DONE
0
E
EDIT
L
$BEGIN
INTEGER A,B;
READ(B);
READ(A);
A:=A+B;
WRITE(A
);
END $
*
```

Figure 4 - Teletype output of an example session with the IHLLDE system. (Underscoring and boxes added)

A outputed. When the user finished his terminal session, the system responded with "PROGRAM DONE." Finally the user called the text editor and gave the command "L"; a listing of his program was printed as shown at the bottom of Figure 4.

The IHLLDE system also has been designed and implemented for an INTEL MCS 80 microprocessor system, and has demonstrated that the microprocessor systems are as well suited for interactive, inexpensive, individual high level language computer systems, as they are for specific uses which require no further programming. It is known that one can learn a programming language faster from an interactive system. The experience of the users of the IHLLDE system supports this conclusion.

4. Selectable Amounts Of Interactivity

An on-line system designed for interactive use should be equally attractive to both experienced and inexperienced users. Because the computer has no method for evaluating the experience level of the user this information would need to be supplied by the user himself. The implication is that the user should be able to modify the man-machine interface during his session with the computer.

Research in the area of interface flexibility clearly indicates that flexibility is not uniformly effective with all users in optimizing performance. In a single encounter with an on-line system, users were more prone to make syntax errors if offered short-cut flexibility options. Nevertheless, almost all users of a flexible version of the system worked significantly faster than those not having the options. The exceptions were the novices who worked more rapidly without the options than with them

[Ref. 2].

The authors' experiences with the Cambridge Monitoring System (CMS) seems to support this conclusion. As a higher level of proficiency in programming, handling the editor and monitor commands and in keyboard entry itself were acquired, previously unannoying responses became increasingly bothersome. For example the "Ready" message response after execution of each CMS command became bothersome, especially when the terminal was communicating with the computer at 110 baud. CMS allows the user to turn this message off.

To efficiently cater to the general population of users the man-machine interface should be designed so that the user would be able to shape the details of the interface for his own convenience.

B. SUPPORT OF STRUCTURED PROGRAMMING

In order to meet the needs of the user and decrease the time to develop reliable, efficient software a system should not only be interactive and flexible but it should also support the use of structured programming.

Structured programming is a technique that embraces the goals of reliability, maintainability and flexibility in software design and implementation [Refs. 5 and 6]. In the initial design stages, structured programming (SP) begins in the form of structured flow charts or pseudo language macros. These in turn consist of and are restricted to a small, well-defined set of program flow control blocks or control functions. The "function modes" of each control block may in turn be composed of other

blocks (see Figure 5). In fact, "top down" strategy starts the initial design with one block, and further refines each function into other blocks until the lowest level of specification is coded. This strategy allows for a maximum integration of segments, modules and programs with a minimum amount of design time. Each functional block has one entry and one exit point which excludes the overlap of functions and increases program reliability.

Along with reliability is the need for readability and ease of debugging. In SP this is enhanced by grouping "chunks" of code (5 to 9 functions limited to 10 to 120 lines of code) into segments which appear on one to three pages of source listing. These segments form a module which in turn forms a program.

Development proceeds top-down and breadth first in SP. All segments of one level are developed in a left to right process, based on sequential order or complexity, before the next level is refined and tested.

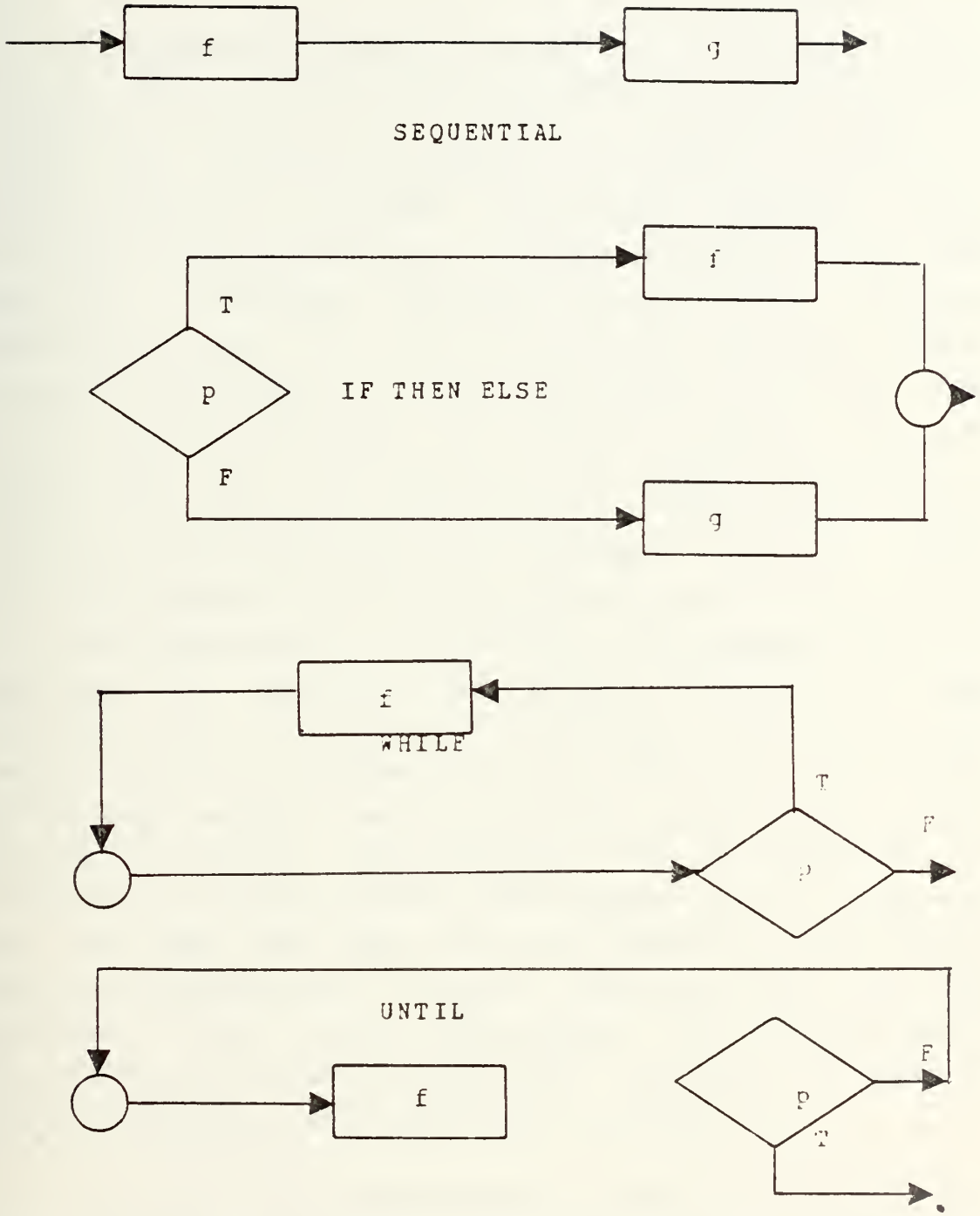

Figure 5 - Basic control structures

1. Macro Processing

An extremely useful and powerful tool that allows for the support of structured programming (especially if the language is assembler) is a macro. A macro is used to extend some underlying language - to perform a translation from one language to another. In assembly language a macro is a means of specifying that a symbol appearing in the code field of a statement actually stands for a group of instructions [Ref. 7]. The use of macros when the user is writing in assembler can substantially ease the user's task in the following ways: a) Often, a small group of instructions must be repeated many times throughout a program with only minor changes for each repetition. Macros can reduce the tedium (and the resultant increased chance for error) associated with these operations. b) If an error in a macro definition is discovered, the program can be corrected by changing the single occurrence of the definition and recompiling/reassembling. If the same routine had been repeated many times throughout the program without using macros, each occurrence would have to be located and changed. Thus debugging time is decreased. c) Duplication of effort between programmers can be reduced. Once the best and most efficient coding of a particular function is discovered, the macro definition can be made available to all other programmers. d) New and useful instructions can easily be simulated. e) Macros assist in program readability and documentation [Ref. 7].

The user of a microcomputer system must often use assembler language as his source language to do his programming. Therefore if the system is to allow for direct support of structured programming it must have a macro capability.

2. Stub Handler

Another feature which the authors felt would complement an interactive development system in support of structured programming is a "stub handler." A stub handler permits the user to define from the keyboard, or from system default memory locations, identifiers and labels which were referenced in the source program but which were not evaluated or defined. An example would be a jump instruction to a label which has not yet appeared in the program. At the point in the code generation phase of assembly when a reference is made to an undefined identifier or label, the user is notified and the terminal keyboard is opened for a definition, allowing code generation to continue with the user supplied value.

The utility of this mechanism becomes more apparent when it is realized that the user may now write his programs initially using calls to modules or subroutines which have not yet been written. This allows the driver portions of the program to assume their final form early on in the development process. System default values for undefined constants could be zero. For undefined subroutines, a call to a location which contains a return instruction would allow execution of the resultant code in most cases. The user is now allowed to convert otherwise non-executable compilations into executable form whether the missing symbolic definition was intentional or accidental.

IV. THEORETICAL BASIS FOR ASID DESIGN

A. FORMAL GRAMMARS AND PARSING

A complete specification of a programming language must perform at least two functions. First, it must specify the syntax of the language. Second, it must specify the semantics of the language; that is, what meaning or intent should be attributed to each syntactically correct program [Ref. 8].

A compiler for a programming language must verify that its input obeys the lexical and syntactic conventions of the language specification. It must also translate its input into an object language in a manner that is consistent with the semantic specification of the language. This translation is referred to as code generation. In addition, if the input contains syntactic errors, the compiler should announce their presence and try to pinpoint their location. To help perform these functions every compiler has a device within it called a parser. Further discussion of parsers requires a review of some basic definitions. The development below generally follows that of Aho and Johnson [Ref. 8].

A grammar is used to define a language and to impose a structure on each sentence in the language. A context-free grammar can often be used to help specify the syntax of a programming language. In addition, if the grammar is designed carefully, much of the semantics of the language

can be related to the rules of the grammar.

In a context-free grammar, two disjoint sets of symbols are used, terminal and nonterminal symbols (sometimes called syntactic categories). In the grammar, one nonterminal symbol is distinguished as the start symbol.

A context-free grammar itself consists of a finite set of rules called productions. A production has the form left-side => right-side, where left-side is a single nonterminal symbol and right-side is a string of zero or more terminal and/or nonterminal symbols. The arrow is simply a special symbol that separates the left and right sides.

A grammar is a rewriting system. If aAb is a string of grammar symbols and A => c, then aAb => acb can be written and it can be said that aAb directly derives acb. A sequence of strings $a_0, a_1, \ldots, a_n$ such that $a_{i-1} \Rightarrow a_i$ for $1 \leq i \leq n$ is a derivation of a_n from a_0. That is a_n is derivable from a_0. For each derivation in a grammar a corresponding derivation tree can be constructed. A derivation tree is a tree whose outermost leaves form a set of terminal symbols in a grammar, whose root is the start symbol, and whose interconnecting nodes form a set of productions of the grammar. Derivation trees are important because they are associated with the parse of a sentence in a grammar.

The start symbol of a grammar, or any string derivable from the start symbol, is a sentential form. A sentential form containing only terminal symbols is said to be a sentence generated by the grammar. The language generated by a grammar G, often denoted by L(G), is the set of

sentences generated by G.

A rightmost derivation is defined to be a derivation in which at each step in the derivation of a sentential form the rightmost nonterminal in each sentential form is rewritten to obtain the next sentential form. Each sentential form derived in this manner is called a right sentential form.

If aAw is a right sentential form in which w is a string of terminal symbols, and aAw => acw, then c is the handle of acw.

A prefix of ac in the right sentential form acw is said to be a viable prefix of the grammar. Restating this definition, a viable prefix of a grammar is any prefix of a right sentential form that does not extend past the right end of a handle in that right sentential form. There is always some string of grammar symbols that can be appended to the end of a viable prefix to obtain a right sentential form. Viable prefixes are important in the construction of left-to-right scanning compilers with good error-detecting capabilities; as long as the portion of the input that has been seen can be derived from a viable prefix, no error has yet occurred.

Frequently, the interest in a grammar is not only in the language it generates, but also in the structure it imposes on the sentence of the language. This is the case because grammatical analysis is closely connected with other processes, such as compilation and translation, and the translation or actions of the other processes are frequently defined in terms of the productions of the grammar.

A parser for a grammar is a device which when presented with an input string, attempts to construct a derivation

tree that matches the input. If the parser can construct such a tree, then it will have verified that the input string is a sentence of the language generated by the grammar. If the input is syntactically incorrect, then the tree construction process will not succeed and the positions at which the process falters can be used to indicate possible error locations [Refs. 8 and 9].

A parser can operate in many different ways. One parser that is efficient for a context-free grammar and well suited for use in compilers for programming languages is an LR parser [Refs. 8, 9, 25].

An LR parser examines the input string from left to right, one symbol at a time. It attempts to construct the derivation tree "bottom-up"; i.e. from the leaves of the derivation tree to the root. An LR parser operates by reconstructing the reverse of a rightmost derivation for the input. This is known as a right parse. An LR(1) parser looks at only the next input symbol before taking an action step.

An LR parser deals with a sequence of partially built trees during its tree construction process. This sequence of trees is referred to as a forest. The forest is constructed from left to right as the input is read.

There are four types of parsing actions that an LR parser can make; shift, reduce, accept (announce completion of parsing), or announce error.

In a shift action, the next input symbol is removed from the input. A new node labeled by this symbol is added to the forest at the right as a new tree by itself.

In a reduce action, a production is specified. A

reduction by a production causes a new node to be created and labeled and the rightmost n roots in the forest (which will have already been labeled) to be made direct descendants of the new node, which then becomes the rightmost tree of the forest.

The parser operates by repeatedly making parsing actions until either an accept or error action occurs.

In order to completely specify an LR parser for a grammar, two tables need to be specified: the parse action table which specifies which actions to take (shift, reduce, accept, or error) with the input symbol depending upon what state the parser is in, and the goto table which specifies which state the parser is to be in for the next parse action.

A properly constructed LR (1) parser can parse a large class of useful languages called the deterministic context-free languages. It has a number of notable properties: (1) It reports error as soon as possible (scanning input from left to right). (2) It parses a string in a time proportional to the length of the string. (3) It requires no rescanning of previously scanned input (backtracking). (4) The parser can be generated mechanically for a wide class of grammars, including all grammars which can be parsed by recursive descent with no backtracking and those grammars parsable by operator precedence techniques.

B. RELATION OF SYNTAX ERROR DETECTION AND CORRECTION TO PARSING

A properly designed LR parser will announce that an error has occurred as soon as there is no way to make a

valid continuation to the input already scanned. Unfortunately, it is not always easy to decide what the parser should do when an error is detected; in general, this depends on the environment in which the parser is operating [Ref. 8]. Any scheme for error recovery must be carefully interfaced with the lexical analysis and code generation phases of compilation, since these operations typically have "side effects" which must be undone before the error can be considered corrected. In addition, a compiler should recover gracefully from each error encountered so that subsequent errors can also be detected.

LR parsers can accommodate a wide variety of error recovery stratagems. In place of each error entry in each state, an error correction routine can be inserted which is prepared to take some extraordinary actions to correct the error [Ref. 8]. Identification of the state frequently provides enough context information to allow for the construction of sophisticated error recovery routines.

Certain automatic error recovery/correction actions are also possible. In particular, the automatic error correction methods described below can be incorporated within an LR parser.

C. AUTOMATIC SYNTAX ERROR DETECTION AND CORRECTION

A very substantial fraction of the time and effort required to develop a program is devoted to the removal of errors. Any compiler should, as much as possible, help the programmer in this chore [Ref. 10].

Early compilers simply rejected programs as soon as an error was detected, vaguely describing the error and where

it was discovered. At the present time, many compilers try to find as many errors as possible. The term error recovery is used to designate the process of determining how to continue analyzing a source program when an error is detected.

Several compilers, most notably the compilers for CORC, CUPL, and PL/C, try to "correct" all errors, generate code and actually execute the program. The term error correction is used to designate the process which, given an incorrect program, transforms it into a correct one. The "goodness" of the process can be measured in some sense by the difference between the corrected program and what the programmer actually meant. Users of error-correcting compilers find it substantially faster and easier to remove errors from programs than with conventional compilers, since, no matter how many syntactic errors, they still have a chance to find logical or run time errors [Ref. 10].

The advantage of error correction over error recovery is twcfold. First, error-correction techniques must be much more precise than error recovery in diagnosis of the error; therefore they provide the programmer with a better description cf his errors. Second, minor errors do not stop a program from executing, and there is a good chance it will be corrected in the right manner [Ref. 10].

Error recovery and error correction are concerned with errors in syntax and in semantics. Logic errors cannot be detected and are therefore not subject to automatic correction. As for semantic errors, only ad hoc recovery techniques exist. Several are described in Gries [Ref. 11].

Misspelling can lead to syntax or semantic errors. When such errors are detected, some compilers try to determine if a spelling error actually occurred. The first work on the

subject is due to Freeman [Ref. 12]. Freeman's algorithm estimates the probability that an identifier is the misspelling of another. Morgan [Ref. 13] has devised a more efficient, but less powerful, method which checks only for the following errors: one letter is wrong, one letter is missing, an extra letter is inserted or two adjacent characters are transposed.

The synopsis below of the most characteristic methods for syntax error recovery is derived from summaries by Levy [Ref. 10] and Graham and Rhodes [Ref. 22].

McKeeman [Ref. 14] describes an admittedly primitve technique similar to techniques used in many bottom up parsers. It uses special characteristics of particular languages. The compiler writer gives a list of "important" symbols, like ";" and "end." When an error is detected, all input symbols are examined and discarded until one is found which is in the list. Then the symbols on the top of the stack are successively examined and discarded until the current input symbol can legally follow what remains of the stack.

For simple precedence parsers [Ref. 15], two papers cover the problem of recovery. In these parsers, errors can be detected in one of two cases: the incoming input symbol is illegal, or the top of the stack does not constitute a phrase.

Wirth [Ref. 16] has a strategy for each case. When the incoming symbol is illegal, a list of "insertion symbols" is scanned. If some symbol of the list is legal between the top of the stack and the incoming symbol, it is inserted. Otherwise the input symbol is stacked. When a reduction cannot take place because the topmost symbols of the stack do not constitute a parse, a table of erroneous

productions is scanned comparing the right parts with the top of the stack. If a match is found, the reduction is performed and the analysis can proceed. The choice of the appropriate insertion symbols and erroneous productions requires a thorough understanding of the analysis algorithm on the part of the compiler designer, as well as a subtle feeling to anticipate frequent misuse of the syntax. Wirth claims that this method yields quite satisfying results. While this technique handles expected errors well, unexpected errors can cause trouble.

Leinus [Ref. 17] approaches the same problem more systematically. The recovery procedure consists of three basic steps where the three-step sequence is executed repeatedly until recovery is complete: (1) Isolate a potential phrase (2) Construct the set of possible "reductions" for the potential phrase (3) Recover by selecting one of the nonterminal symbols in the set to replace the phrase; if the selection attempt fails, repeat from step (1). The actual process is complex. Leinus does not thoroughly justify the choice of his algorithm, but it has the merit of being systematic.

Five more heuristic methods exist, none of which make use of special features of a particular language.

Gries's scheme [Ref. 11] works for bottom-up parsers such that an error is detected when the input symbol is illegal to follow the stack (for bottom-up parses, the stack contains the head of the sentential form). It tries, whenever an error is detected, to insert a substring in front of the current input symbol, such that the substring is legal in the context constituted by some stack symbols at the right-hand side and by the input symbol at the left. If no such substring exists, the current input symbol is discarded and the process repeated with the next input

symbol. This technique has been successfully used in an intuitive manner for error recovery in a compiler using transition matrices
[Ref. 18]. The problem with this technique is that it requires a substantial amount of programming effort for the error recovery portion of the compiler. Furthermore, although such a method handles the expected errors reasonably well, it can fail badly on unanticipated errors.

Irons [Ref. 19] has developed an error-recovery method for a top-down parser. A top-down parser constructs a derivation tree starting with the top node, or start symbol and successively adds lower branches and nodes. In order to avoid backup, Iron's parsing algorithm constructs several candidate syntax trees in parallel. At any step during the parse, one or more trees have been constructed; some branches are incomplete. An error is detected when no partial tree can be built further. Then all input symbols are successively examined and discarded until one is found which is a potential node of some incomplete branch. A terminal string is determined such that, if inserted before this input symbol, the continuation of the parse would cause this symbol to be correctly linked to the incomplete branch. The string is inserted and the parse continues. Irons's technique uses much more context than that of Gries because the parse is top-down and contextual information is easy to extract from the incomplete trees.

LaFrance [Refs. 20 and 21] describes a recovery technique for parsers using Floyd Production Language. When an error is detected in a state where only one next action is possible, this action is taken. Otherwise, a set of intuitive and predetermined rules for transforming the top of the stack and a fixed number of subsequent symbols is used. Which rule to apply is determined by comparing the actual symbols with the set of symbols which "could legally

be there." The process looks for a match according to a predetermined set of patterns. With each pattern is associated a transformation. For example, if the current input string is abcd and if bacd is legal in the current context, then 'a' and'b' are transposed in the input string. Thus, this process performs transformations which are more complex than those performed by the methods of Gries and Irons. The problem with this approach is that if multiple parsing continues for an unbounded number of steps, an explosion in space and time ensues. LaFrance bounds the amount of multiplicity. This improves efficiency, but can yield insufficient information in some cases.

Levy [Ref. 11] describes a model for error correction for formal languages which have one-way deterministic acceptors. This process makes "local" corrections over clusters of errors, using the context around the errors to determine the correction and to insure that the different local corrections performed on the string do not interfere with one another. The error-correction process is embedded in left-to-right recognizers. The parsing of correct strings is not slowed down by the presence of the error-correction mechanism. This mechanism uses the recognizer both to detect errors and to find possible corrections.

Levy has attempted to find a theoretical basis for error correction in all deterministic context-free parsing methods having the correct prefix property. His method includes a backward move on the input to determine the entire left context of the error discovery point that could contain the error and then parallel parses from the beginning of the left context to pursue all possible minimal distance corrections of a fixed bound distance. This method has the same problem as LaFrance's method. Levy has proposed some heuristics to improve its efficiency.

Graham and Rhodes [Ref. 22] describe an error recovery method which can be incorporated in any bottom up parser which does nct back up. The error recovery routines are invoked when a syntax error is detected by the parser. Control is returned to the parser when the error state has been removed. The method attempts to analyze the context in which the error occurs. There are two phases in the method, a condensation phase and a correction phase.

In the condensation phase, an attempt is first made to make further reductions on the stack, preceding the point of error detection. This attempt is a "backward" move. A "forward" move is an attempt to parse the input just beyond the point of error detection. The forward move terminates either because a second error is detected further on in the input, or more likely, because the only possible next parsing action is a reduction involving that part of the stack containing the detected error. The forward and backward moves are an attempt to summarize the context surrounding the point at which the error was detected.

The correction phase considers changes to sequences of symbols, rather than isolated changes to simple symbols, so that as much of the context that surrounds the error can be efficiently exploited. The quality of recovery can be traded for efficiency in choosing a correction. The idea is to change the parsing stack, at the point of error, to a right-hand side of a production of the grammar or to one or more prefixes of right-hand sides which "fit in" in the sense that they can legitimately occur in the given context. In general, there usually is more than one possible change that appears locally to correct the error. To increase the likelihood that the change really corrects the error and to provide helpful diagnostic information to the programmer an effort is made to choose the "best" correction. This is accomplished by determining which of the possible locally

correct changes requires a minimum of symbol by symbol modification of the parsing stack.

The Graham-Rhodes method has been empirically tested against the Wirth method and the method used on the PL/C compilers and it appears to be qualitatively better than either the Wirth method or the PL/C compiler method.

Hopcroft and Ullman [Ref. 23], and Smith [Ref. 24] have studied formal error correction. The papers are highly theoretical, examine only the very specific case where errors are just substitution of symbols, and their mechanisms are very complex and time consuming. In particular, the time needed to parse a correct string is considerably greater than if a usual parser is utilized.

V. ASID DESIGN

A. PURPOSE

The user can best be aided by the computer during the software development process if the man-machine interface is such that the user can obtain immediate results from his endeavors. In most systems the user writes his program or a significant portion of a program and then submits his code to a language processor to see if it is syntactically correct. If the code segment was syntactically correct and the code segment has all the necessary semantic parts to make it a program, the user would then attempt execution of his code.

The authors' contention was that the user would be better served during the initial entry of a program by directing the computer to analyse small portions of his code for syntax immediately as each segment is input. If a syntax error is detected, the user is informed immediately and may make the necessary changes before continuing. After syntactic analysis the user's input could be executed. For this to be done a segment must be defined as some arbitrarily small element of the source language. This segment is determined by the grammar which is used to define the language in which the user is writing. In the CAPS system each character is analysed as it is introduced into the input stream. Thus the user is constantly assured that his input up to the last segment is some valid prefix of a source language program.

A more powerful extension of the immediate parse would be immediate execution of the source program as executable segments appear in the input stream, with the results of each such execution being displayed to the user. At this point the user knows that his code is syntactically correct and he has seen a detailed view of its execution to assist him in detecting errors in the logic of his program.

The computer would now be much more able to assist the user in producing a running and logically correct program on the first attempt.

As a realization of the need previously expressed for a software development system which could aid the typical user to the fullest extent the authors have designed a support program named ASID.

ASID accepts INTEL 8080 assembler language as defined by the INTEL Corporation [Ref. 7] and emits machine code for the 8080 microprocessor. ASID is written in PL/M and runs in consonance with the CP/M operating system [Ref. 30] on an 8080 based microcomputer with at least 20 kilobytes of memory and an auxiliary storage device capable of random access of records of stored data.

B. MAJOR FUNCTIONAL BLOCKS

The basic structure of ASID is that of a two-pass assembler as presented in Barron [Ref. 29]. Figure 6 shows the working relationships between the classical functions of a two-pass assembler and shows the integration of the monitor, stub handler, error module, and execution monitor.

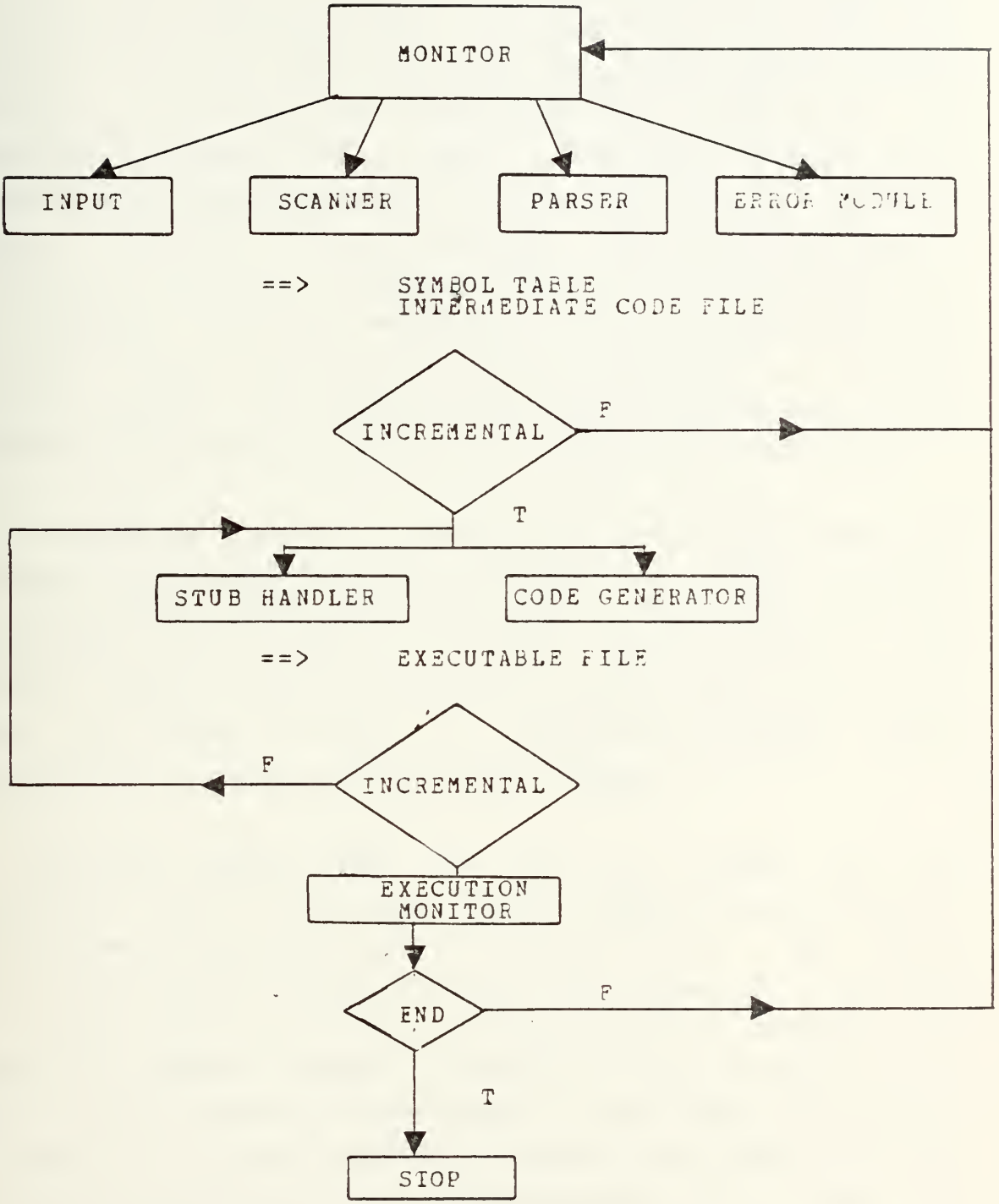

Figure 6 - ASID system diagram

The desirability of a stub handler was expressed earlier. The user may select to toggle the stub handler on or off. If the stub handler toggle is off and a reference is made to an undefined label or identifier, the run is aborted. If the toggle is on then the stub handling module of ASID is called from the code generating portion of the assembler's second pass when a reference is made to an identifier or label which is not defined in the program segment currently being processed. When the stub handler toggle is on then two sources of definition are available to the program. One is selected constant values which are provided by ASID. The other source is the user. The stub handler prints the name of the identifier or label on the terminal and opens the keyboard for input of a definition.

The monitor function is responsible for initializing the program and allowing the user to change the system toggles at any time. This allows the user to amend the amount of interaction and the type of products ASID provides. The monitor function also includes the looping mechanisms which cause the parsing action to immediately process each sentence as it appears in the input stream.

The error handler, when the user has opted for the incremental mode of operation, causes immediate notification to the user of the detection of a syntax error. The token (or in the case of a scanner error, the character) which was not a valid input is shown to the user and the keyboard is opened to accept another attempt at a well-formed input line. In the noninteractive assembly mode, the entire line in which the error occurred is deleted and ASID execution continues to parse and check for syntax but no attempt is made to generate executable code.

The execution monitor is used in incremental mode only

and performs a controlled execution of the source program, line by line as each line completes a successful parse action. The state of the machine as the user has formed it is copied and restored at the beginning and end of each instruction execution. The stub handler provides the mechanism to patch up forward references.

Batch operation utilizing a previously created file is also possible with or without the incremental feature.

Those instructions which cause an alteration of the program counter or which use register M as an operand must, in the present system, be interpreted rather than actually executed as their execution would most likely result in alteration of a portion of ASID system code or data area. Although this is not true execution of the user's source code in the pure sense, it does provide the user with the opportunity to view the results of all register and accumulator operations as an aid to detection of logic errors.

At the completion of initial program entry, a conventional code generation pass is run on the user's program, creating an executable file.

C. BASIC SYSTEM OPERATION

The dialect of 8080 assembler language which is processed by ASID is described in a formal grammar in Appendix A . The input stream is scanned for tokens by the scanner module which is driven by a state transition matrix. The tokens built by the scanner are processed by the parser section which operates from a set of tables generated by the LALR(1) parser algorithm [Ref. 25]. Both the scanner and

the parser are able to detect syntactical errors in the input. All error recovery action is initiated from the parser.

Most assemblers form the actual machine code words for those assembler mnemonics which have a register or two registers as operands through some regular mathematical combination of a base value (unique for each instruction type) and the numerical value associated with each register designation. Such regular combinations are built into the inner workings of the CPU. The INTEL 8080 is no exception to this pattern. By following such a scheme, "MOV M,M" produces a "HLT" instruction. Also, "LXI A,_" would produce an "LXI SP, _." The grammar for ASID does not allow the parser to recognize such constructions. The parser signals a syntax error if they appear in the input stream.

Four files are maintained by the ASID system during program creation. These files are a source file, a macro reference file, an intermediate code file and an executable code file. Although ASID does not incorporate a text editor, any files created by an ASID user can be edited by the CP/M Context Editor [Ref. 31]. The editing functions which are available to the user while creating a program with ASID are those console line editing functions provided by CP/M. Specifically, the user is limited to removing the last character or removing the entire line.

Macros are processed by ASID in line with the source program. The requirement for the user is that he define all macros prior to their use in his program. A call to an undefined macro will cause the ASID session to abort. Presently, macro calls may not be used inside of macro definitions. Such use of a macro will cause ASID to enter an undetermined state. There is effectively no limit to the number of formal parameters allowed with a macro definition.

Formal parameters have "scope" only within their declared macro body.

The user may gain access to the monitoring routines in ASID to alter any or all of the system toggles whenever the terminal keyboard is open for input. The user establishes contact with the monitoring section by typing the attention character "!" immediately followed by the toggle number and the value to which the toggle is to be set. Several toggles may be set on the same command line, each toggle reference separated from the next by a comma. A semicolon must terminate an attention line.

Input to ASID is free form. Embedded blanks are important as delimiters. Any number of blank spaces is treated as if it were just one blank. Comments may be freely inserted anywhere in the program text between the beginning and ending comment delimiters "< ... >." Each ASID statement must be terminated by a semicolon. Multiple statements may appear on one line, and statements may be continued to the next line.

If the user has opted for incremental mode operation of ASID, each sentence of source code is scanned, parsed, converted to intermediate code form and executed. This execution step may involve the stub handler if the source code sentence used an identifier or label which is a forward reference. This process will continue until the psuedo operation code "END" is processed. END causes the entire intermediate file to be passed through the code generation phase to build a complete executable file.

Upon completion of incremental execution of a line of code, the user may access the copy of the CPU for his machine and alter the values of the registers. Access is gained by entering "$" followed by the values the user

desires to insert. The order of input is significant, inputs must be separated by commas and an input must be present for each register pair. Ordering of inputs is as follows: A and Flags, B and C, D and E, H and L. All inputs must be in hexadecimal radix and the line must terminate with a semicolon.

VI. ASID IMPLEMENTATION

A. BASIC ASSEMBLER FUNCTIONS

1. Scanner

The scanner is driven by a state transition matrix. A state transition matrix requires two entering arguments, the current state of the process and the next item of information to be processed. These two arguments typically correspond to the rows and columns of a two dimensional linear array. The data items in the array are used to designate a specific action from a list of possible action to be executed or accomplished.

One state must be designated as the initial state from which all further processing will take place. Once the initial state has been designated the processing can proceed.

Primitives used by the scanner deliver the input stream one character at a time. Each character delivered is used as the column index into the state transition matrix. The row index is determined by the current state of the process. The resulting data item extracted from the state transition matrix is used as an index into a CASE statement which will take one or a combination of several actions. These actions include adding the character to a 32 byte array used as an accumulator, modifying the current state of

the process, recognizing and reading through comments, recognizing the monitor commands, detecting and reporting malformed tokens, and recognizing tokens.

The scanner has the ability to recognize and deliver directly to the parser "special character" tokens, "string" data items and "number" tokens. All numbers are converted to binary form before parsing action is initiated. All other alphanumeric strings (identifiers) that the scanner recognizes are placed in the accumulator. The string in the accumulator also has had a hashing value computed. When the scanner has determined it has a valid token it delivers the token to the parser and resets the current state of the scanner process to the initial state.

The scanner was implemented as a state transition machine because the authors felt it would be fast, easily modified and easily understood.

The speed advantage is realized due to the fact that the next action to perform with a particular input character is not determined by a series of conditional tests but rather a straight-forward index into a matrix and a CASE statement.

During the development of ASID the authors noted that the actions to be taken with a specific character could be changed easily by just altering the entry in the state transition matrix, thus alleviating the need for a major change in the code of the scanner module itself. This characteristic saved much development time and allowed corrections to the original logic design to be easily implemented.

2. Symbol Table

The symbol table stores attributes of program entities such as identifiers, labels and macro names. The information stored in the symbol table is built and referenced by the assembler. The symbol table data structure is a declared linear array whose first 780 bytes are initialized to contain the reserved word list.

The user portion of the symbol table is an unordered linear list of entries. Individual elements are accessed through a chained hash addressing scheme. Each entry in the hash table heads a linked list whose printnames all evaluate to the same hash address. A zero in the hash table indicates no entries exist on that particular chain. During references to the symbol table the accumulator, the global variable ACCUM, contains the string of ASCII characters which comprise the token with the first byte being the length of the printname. The global variable HASH$SUM is set to the sum modulo 128 of the ASCII characters in the printname. Entries are chained in the order of their occurrence in the program.

Each entry in the symbol table consists of a variable-length vector of four entries. These are attribute, collision pointer, value and printname. The first item in the printname field is the length of the string. The maximum length of a printname in the symbol table is eight characters, although 31 characters will be accepted by the scanner. When a search of the symbol table is being conducted, the first items to be compared are the respective length fields. Only if the lengths are the same will a character by character match be attempted.

Macro names have the head-of-chain pointer for their parameter lists stored in the value field. When a macro expansion is being processed, all searches for tokens begin with the parameter list chain. If the item is not found then the symbol table is entered at the global level and the search continues. This method establishes local scope for macro formal parameters.

The first 780 bytes of the symbol table contain all the assembly language mnemonics. These entries appear in the hash list in the same manner as user defined identifiers. The reserved word entries contain, in the attribute word, information concerning whether this is a psuedo operation or an actual assembler instruction, if an assembler instruction how many data bytes will follow the word of machine code, the token number for that type of instruction for information to the parser, and the method which must be used to incorporate any register operands into the machine code word. The value field of a reserved word is one byte and contains the basic value needed to compute the actual machine code word corresponding to that instruction. The collision pointer field and printname field are identical to user defined items, as all elements present in the symbol table are searched in the same manner.

When designing ASID the authors decided to use a declared linear array with subscripts as pointers rather than "based variables and pointers" which are also supported by PL/M. This choice was made to increase the readability of the ASID source program.

With the reserved word list and the symbol table combined as one data structure, only one set of procedures was needed to conduct all searching and information extraction functions.

3. Parser

The parser used by ASID is a table-driven pushdown automaton of the LR(1) type described in section IV A. It receives a stream of tokens from the scanner and analyzes them to determine if they form a sentence of the 8080 assembler language. The 8080 assembler grammar is designed so that each statement parses to a complete sentence, causing a source program to appear as a series of sentences. When an error is detected and ASID is in the incremental mode, the parser gives an error message to the user and tells him with which token the error was discovered. The sentence in error is deleted and the parser reinitialized. The user is then allowed to reenter the line which was in error with a correction. In the nonincremental mode the line is simply deleted, the parser reinitialized, and the next sentence ("program") parsed. The major data structures in the parser are the parse tables and the parse stack.

The reasons an LR(1) parser was used in the ASID system were its efficient operation with a context-free grammar, its error detecting capabilities,its ability to accommodate a wide range of error recovery/correction strategies, and its ability to be generated easily and mechanically for the grammar the authors used to describe the 8080 assembler language. The one disadvantage of using an LR(1) parser with the ASID system is that it requires a large amount of memory.

4. Pass 2, Code Generation

In addition to verifying the syntax of source statements, the parser also acts as a transducer by

associating semantic actions with reductions. Each time the parser determines that a reduction should take place, the procedure CODER is called with the production number as a parameter. Some productions have no semantic actions associated with them. Various global variables contain pertinent information concerning the previously parsed tokens. This information is used by the various production actions to manipulate the symbol table, build an intermediate code file and manipulate the macro reference file.

Because macros and the psuedo operation SET and EQU are processed in-line during pass one, several logical flags are needed to coordinate the actions of the parser and the CODER procedure. These flags control the redirection of program products and source to and from the macro file.

If the user has opted for incremental mode, then as each sentence of source completes parsing (accept state is reached) the intermediate code for that sentence is passed to the executable code generator. Procedure EXGEN determines whether or not the last sentence was an assembler instruction or a psuedo operation. Assembler instructions are represented by the appropriate eight bit machine code word. If a data value or address is needed to complete an instruction, the expression is evaluated and the symbol table is consulted as needed. If the symbol table entry indicates that this item is undefined (has not been given a value) the stub handler is called.

When the CODER sees the psuedo operation END it signals the assembler to reset its input back to the beginning of the intermediate code file and pass the entire program through EXGEN. The result is the executable version of the user's program. The stub handler is available during this portion of ASID execution.

B. MACRO PROCESSING

To more fully support the user of 8080 assembler language, ASID possess a macro processor. The macro processor employed by ASID is an in-line processor, rather than a separate preprocessor. This was made possible by including the macro definition and macro call in the formal grammar. When the parser is in the process of verifying input and it discovers a macro definition or macro call then certain logical flags are set to facilitate the redirection of the intermediate code generated by procedure CODER.

When a macro definition is being processed, the processing of the macro body is handled in the same manner as non-macro statements with the exception that the intermediate code is directed to the macro reference file, and no incremental execution is performed. This redirection of intermediate code terminates when the psuedo operation ENDM is seen. ENDM closes the macro file for the definition being processed and redirects the intermediate code back to the intermediate file associated with the main body of the program.

When a macro call is recognized, the macro reference file is searched to locate the corresponding macro definition. If the search is successful, the corresponding macro definition, which has already been converted to intermediate code form, is copied into the main program intermediate code file. If ASID is in incremental mode, each sentence of the macro definition will be executed incrementally as it is inserted in the main program intermediate file. If the macro definition is not found a fatal error occurs and the ASID session is aborted.

C. ERROR MODULE

The current error module is very crude in the ASID system. It is called by the parser when a syntax error is detected. Its major function is to undo what has been done by the CODER and symbol table manager up to the point in the parse where the error was discovered. This consists mainly of resetting pointers and flags back to their proper values so that a graceful recovery from the error can be accomplished and further errors can be detected. If the system is in the incremental mode, "error snowballing" is eliminated as the user is asked to correct the error before another parse with a different statement is done. As mentioned in the parser section of ASID implementation and in section IV B, an error module that is capable of automatic error recovery/correction would certainly be feasible and desirable in the ASID system. The main disadvantage of an automatic error recovery/correction module would be the amount of memory needed to incorporate it in the system.

D. INCREMENTAL EXECUTION MONITOR

The incremental execution monitor is operative only if the user has opted to employ ASID in the incremental mode. When enabled, this portion of ASID causes an actual execution of the latest line of the user's program. Certain instructions, however, cannot be allowed to execute because only one line of the user's object code is loaded into the execution buffer. If a jump or call instruction were actually executed, control of the computer would be lost.

At the end of each execution cycle the contents of the CPU registers may be displayed at the terminal. The user may then check out the logic of his program in some detail before entering the next line.

During incremental execution no attempt is made to preserve previously executed instructions or user defined data areas. Consequently, instructions which contain references to register M invoke the stub handler prior to execution. All information stored in the symbol table is available during incremental execution.

If it is determined that the current instruction can be incrementally executed then the machine instruction word and associated data byte or bytes areloaded into a four-byte array in memory. A RST 1 instruction is executed, the CPU registers are copied into memory and the contents of the registers the way the user left them at the end of his last execution is placed into the CPU. The program counter is next set to the address of the user's instruction. The two bytes set aside for user instruction data were initialized to zero (NOP instructions). The fourth byte in the user execution buffer is a RST 2 instruction which invokes a routine which copies the user's machine state into memory and restores ASID.

If the user has requested the display of registers then ASID causes his machine state to be printed at the terminal. ASID is now ready to accept the next line of input.

It might seem that there have been rather severe limitations placed on which instructions can be incrementally executed. The important thing to be remembered is that at this point in program creation with most other systems, all the user has done is to create a portion of a source file or punch a deck of cards. He has

no information whatsoever concerning the correctness of his syntax let alone what the program will do when it is executed. Using ASID, however, the user is assured of correct syntax and has seen the results of controlled execution of a portion of his program.

VII. SYSTEM EVALUATION

A. AREAS OF APPLICATION

The authors felt that ASID would have a high degree of applicability in the educational environment. The time taken to learn any programming language is significantly reduced when an interactive system capable of immediate feedback is utilized by the student. When contrasting assembly language to a higher level language the user of assembly language is not bound by detailed rules for structuring segments of code. This attribute of assembly language creates an environment in which it is difficult for the user to continually co-ordinate the individual steps involved in his program with the overall desired result. ASID allows the user to receive immediate feedback concerning the actual results of execution of each instruction as it is entered. This enables a student user to verify that his instructions are causing the operation which he intended. ASID also eliminates such troublesome instruction constructs as MOV M,M and LXI A, _.

The experienced user can also benefit from ASID because it was designed to support structured programming techniques and should reduce initial program entry/assembly time.

B. EXTENSIONS

ASID in its present form is a complete system. There are certain additional features and functions which the authors feel would enhance the operation of the system.

A "mini" text editor would be advantageous. This would allow the user to correct only the token which is in error rather than the present convention of requiring the entire line to be re-entered. Insertion and deletion of tokens would also be allowed.

The incremental execution monitor could be expanded in function to include the building of a memory map of the user's program as it is being processed in incremental mode. This would enable the controlled execution of the presently troublesome JUMP, CALL and memory reference instructions as long as the references and labels are defined in the program prior to their use as an operand. Forward referencing could still be the responsibility of the stub handler.

The error module could be expanded to include one of the automatic syntax error correction schemes presented in section IV C. The authors specifically recommend the method developed by Graham and Rhodes [Ref. 22]. This method is designed to be easily interfaced with an LR(1) parser.

The facility for modifying the user's CPU is somewhat cumbersome in its present form. An extentson could be to provide selective modification of one or more of the registers.

Another extension pertains to the macro processor.

Specifically, ASID in its present form, does not allow the nesting of macro definitions nor does it allow macro calls within macro definitions. Both of these features would greatly enhance the overall utility of the system.

Presently ASID does not provide a cross-reference listing of user defined variable names, nor does the source code file receive an automatic formatting treatment, whereby label, instruction and operand fields are aligned using preset tab positions. Both of these features would be useful to the user. A facility that allows the user to specify a library routine and have it linked to his program is another extension that the authors feel would be of great value to the user.

VIII. SUMMARY AND RECOMMENDATIONS

ASID was an attempt by the authors to integrate state of the art compiler writing techniques, a highly interactive environment and an 8080 based microcomputer mainframe. The resultant system processes 8080 assembler language incrementally with each complete line or sentence of assembler code being the unit of interaction. In addition, ASID is capable of performing a controlled execution of each instruction after a successful assembly of the line. A macro processor was included in ASID which processes macro definitions and macro calls in-line on the first pass of the assembly.

The authors have concluded that:

1) Assembly language for the 8080 can be described by a context-free grammar which is recognized and accepted by an LALR-1 parser.
2) Incremental compilation during initial program entry on a dedicated hardware system is feasible. No undue delays or interruptions were perceived by the user during program entry.
3) Macros can be processed effectively in-line.
4) Incremental execution is feasible.

The present form of ASID, although runnable, is not appropriate for public distribution. The source for ASID is available through the Chairman of the Department of Computer Science, Naval Postgraduate School, Monterey, California 93940. Several areas for further extensions of the system have been mentioned and the authors feel that ASID is well suited as the basis for further research, particularly in

the areas of error recovery and incremental execution.

APPENDIX A

FORMAL GRAMMAR for ASID

```
<START>                    ::= <PRIMARY INSTRUCTION>
<PRIMARY INSTRUCTION>      ::= <LABELED INSTRUCTION>
                               <MACRO DEFINITION>
                               <MACRO CALL>
                               <BASIC INSTRUCTION>
<LABELED INSTRUCTION>      ::= <LABEL>
                               <LABEL> <BASIC INSTRUCTION>
<BASIC INSTRUCTION>        ::= <ASSEMBLER MNEUMONIC>
<ASSEMBLER MNEUMONIC>      ::= <MNON ZERO>
                               <MNON ONER><REGISTERM>
                               <MNON ONER><REGISTER1>
                               <MNON ONER> <REGISTERPS>
                               <MNON ONER> <REGISTERH>
                               <MNON ONERP><REGISTERPS>
                               <MNON ONERP><REGISTERH>
                               <MNON ONERP><REGISTERSP
                               <MNON ONEST><REGISTERPS>
                               <MNON ONEST><REGISTERH>
                               <MNON ONEST><REGISTERPSW>
                               <MNON ONEA><EXPRESSION>
                               <MNON ONEADEFN><DLIST>
                               <MNON ONEB><EXPRESSION>
                               <MNON ONEBRST><EXPRESSION>
                               <MNON ONELS><REGISTERPS>
                               <LEFT PART><EXPRESSION>
                               <MNON TWOMV><REGISTERM> ,
                               <REGISTER1>
                               <MNON TWOMV><REGISTERM> ,
```

```
                                <REGISTERPS>

                                <MNON TWOMV><REGISTERM> ,
                                <REGISTERH>

                                <MNON TWOMV><REGISTER1> ,
                                <REGISTERM>

                                <MNON TWOMV><REGISTERH> ,
                                <REGISTERM>

                                <MNON TWOMV><REGISTERPS> ,
                                <REGISTERM>

                                <MNON TWOMV><REGISTER1> ,
                                <REGISTER1>

                                <MNON TWOMV><REGISTERPS> ,
                                <REGISTERPS>

                                <MNON TWOMV><REGISTERH> ,
                                <REGISTERH>

                                <MNON TWOMV><REGISTERPS> ,
                                <REGISTER1>

                                <MNON TWOMV><REGISTER1> ,
                                <REGISTERPS>

                                <MNON TWOMV><REGISTERPS> ,
                                <REGISTERH>

                                <MNON TWOMV><REGISTERH> ,
                                <REGISTERPS>

                                <MNON TWOMV><REGISTERH> ,
                                <REGISTER1>

                                <MNON TWOMV><REGISTER1> ,
                                <REGISTERH>

                                <MNON TWOMI><REGISTER1> ,
                                <EXPRESSION>

                                <MNON TWOMI><REGISTERPS> ,
                                <EXPRESSION>

                                <MNON TWOMI><REGISTERH> ,
                                <EXPRESSION>

                                <MNON TWOMI><REGISTERM> ,
                                <EXPRESSION>

                                <MNON TWOLI><REGISTERPS> ,
                                <EXPRESSION>

                                <MNON TWOLI><REGISTERH> ,
                                <EXPRESSION>

                                <MNON TWOLI><REGISTERSP> ,
                                <EXPRESSION>

<LEFT PART>             ::=  <IDENTIFIER> EQU

                             <IDENTIFIER> SET

<MACRO DEFINITION>      ::=  <IDENTIFIER> MACRO

                             <IDENTIFIER> MACRO <PLIST>

<MACRO CALL>            ::=  <MACRO&>
```

```
                                    <MACRO&> <PARM LIST>
<REGISTER1>                 ::=     A
                                    C
                                    E
                                    L
<REGISTERPS>                ::=     B
                                    D
<REGISTERSP>                ::=     S
<REGISTERPSW>               ::=     P
<REGISTERH>                 ::=     H
<REGISTERM>                 ::=     M
<DLIST>                     ::=     <EXPRESSION>
<EXPRESSION>                ::=     <LOGICAL TERM>
                                    <EXPRESSION> OR
                                    <LOGICAL TERM>
                                    <EXPRESSION> XOR
                                    <LOGICAL TERM>
<LOGICAL TERM>              ::=     <LOGICAL FACTOR>
                                    <LOGICAL TERM> AND
                                    <LOGICAL FACTOR>
<LOGICAL FACTOR>            ::=     <LOGICAL PRIMARY>
                                    NOT <LOGICAL PRIMARY>
<LOGICAL PRIMARY>           ::=     <ARITHMETIC EXPRESSION>
<ARITHMETIC EXPRESSION>     ::=     <TERM>
                                    <ARITHMETIC EXPRESSION> +
                                    <TERM>
                                    <ARITHMETIC EXPRESSION> -
                                    <TERM>
                                    - <TERM>
                                    + <TERM>
<TERM>                      ::=     <PRIMARY>
                                    <TERM> * <PRIMARY>
                                    <TERM> / <PRIMARY>
                                    <TERM> % <PRIMARY>
                                    <TERM> MOD <PRIMARY>
                                    <TERM> SHL <PRIMARY>
                                    <TERM> SHR <PRIMARY>
<PRIMARY>                   ::=     <IDENTIFIER>
```

```
                                <NUMBER>
                                ( <EXPRESSION> )
                                <STRING>
                                <EQU NAME>
                                <SET NAME>
                                <DEFINED LABEL>
<PLIST>                 ::=     <IDENTIFIER>
                                <PLIST> , <IDENTIFIER>
<PARM LIST>             ::=     <OPERAND>
                                <OPERAND> , <PARM LIST>
<OPERAND>               ::=     <REGISTER1>
                                <REGISTERPS>
                                <REGISTERH>
                                <REGISTERM>
                                <REGISTERSP>
                                <REGISTERPSW>
                                <EXPRESSION>
```

LIST OF REFERENCES

1. Endres,A., "An Analysis of Errors and Their Causes," IEEE Transactions on Software Engineering, v. SE-1, p. 140-149, June 1975.

2. Walther, G.H. and O'Neil, H.F., Jr., "On-Line User-Computer Interface: The Effects of Interface Flexibility, Terminal Type and Experience on Performance," National Computer Conference, v. 43, p. 379-384, 1974.

3. Chu, Y. and Cannon, E.R., "Interactive High-Level Language Direct-Execution Microprocessor System," IEEE Transactions on Software Engineering, v. SE-2, no. 2, p. 126-134, June 1976.

4. Wilcox,T.R., Davis, A.M. and Tindall, M.H., "The Design and Implementation of a Table Driven, Interactive Diagnostic Programming System," Communications of the ACM, v. 19, p. 609-616, November 1976.

5. Rieks, G.E., "Structured Programming in Assembler Language," Datamation, v. 22, p. 79-84, July 1976.

6. Herman-Giddens, G.S., Warren, R.B., Barr, R.C. and Spach, M.S., "Biomac: Block Structured Programming Using PDP-11 Assembler Language," Software-Practice and Experience, v. 5, p. 359-374, 1975.

7. Intel Corporation , Intel 8080 Assembly Language Programming Manual, 1976 .

8. Aho, A.V. and Johnson, S.C., "LR Parsing," Computing Surveys, v. 6, p. 99-124, June 1974.

9. Aho, A.V. and Ullman, J.D., The Theory of Parsing, Translation, and Compiling Volumes I and II, Prentice-Hall, 1973 .

10. Levy, J., Automatic Correction of Syntax Errors in Programming Languages, Ph.D. Thesis, Cornell University, 1971 .

11. Gries, D, Compiler Construction for Digital Computers, Wiley, 1971 .

12. Freeman, D.N., Error Correction in CORC: the Cornell Computing Language, Ph.D. Thesis, Cornell University, 1963 .

13. Morgan, H.L., "Spelling Correction in System Programs," Communications of the ACM, v. 13, p. 90-94, February 1970.

14. McKemman, W.M., Horning, J.J. and Wortman, D.B., A Compiler Generator, Prentice Hall, 1970 .

15. Wirth, N. and Weber, H., "Euler, a Generalization of ALGOL and its Formal Definition," Communications of the ACM, v. 9, p. 13-23 and 89-99, January and February 1966.

16. Wirth, N., "A Programming Language for the 360 Computers," Journal of the ACM, v. 15, p. 37-74, January 1968.

17. Leinus, R.P., Error Detection and Recovery for Syntax Directed Compiler Systems, Ph.D. Thesis, University of Wisconsin, 1970 .

18. Gries, D., Paul, M. and Wiehle, H.R., "Some Techniques Used in the ALCOR ILLINOIS 7090," Communications of the ACM, v. 8, p. 496-500, August 1965.

19. Irons, E.T., "An Error-Correcting Parse Algorithm," Communications of the ACM, v. 6, p. 669-673, November 1963.

20. LaFrance, J.E., "Optimisation of Error-recovery in Syntax Directed Parsing Algorithms," SIGPLAN Notices, v. 5, p. 2-17, December 1970.

21. LaFrance, J.E., Syntax-Directed Error Recovery for Compilers, University of Illinois, Department of Computer Science, Report 459, June 1971 .

22. Graham, S.L. and Rhodes, S.P., "Practical Syntactic Error Recovery," communications of the ACM, v. 18, p. 639-650, November 1975.

23. Hopcraft, J.E. and Ullman, J.D., Error Correction for Formal Languages, Princeton University, Department of Electrical Engineering, Technical Report 52, November 1966 .

24. Smith, W.B., "Error Detection in Formal Languages," Journal of Computer and System Sciences, v. 4, p. 385-405, October 1970.

25. Lalonde, W.R., An Efficient LALR Parser Generator, Ph.D. Thesis, University of Toronto, 1970 .

26. Berthaud, M. and Griffiths, M., "Incremental Compilation and Conversational Interpretation," Annual Review in Automatic Programming, Pergamon Press, v. 7, p. 95-114, 1974.

27. Martin, J., Design of Man-Computer Dialogues, Prentice-Hall, 1973 .

28. Tindall, M.H., An Interactive Compile-Time Diagnostic System, Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1975 .

29. Barron, D.W., Assemblers and Loaders, Macdonald-Elsevier, 1969 .

30. Kildall, G.A., An Introduction to CP/M Features and Facilities, Digital Research, 1976 .

31. Kildall, G.A., ED: A Context Editor for the CP/M Disk System, Digital Research, 1975 .

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	2
4.	Asst Professor V. M. Powers, Code 52Pw Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
5.	LCDR S. T. Holl, USN, Code 52Hl Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
6.	LT John L. Cuzzocrea, USN Class 57 SWOSCOLCOM BLG 446 Newport, Rhode Island 02840	1
7.	LT Michael C. Thomas, USN Class 57 SWOSCOLCOM BLG 446 Newport, Rhode Island 02840	1

Thesis 171408
C974 Cuzzocrea
c.1 An interactive, incremental assembly language processor for the INTEL 8080.